10/19/2009

JBoss Cluster, EJB Timers, and Singletons, Oh My...

JEE provides a lot of things, but there are some basic CS concepts that are difficult to implement simply (or at all). Queues come to mind... I need a simple queue. One queue please, Vassili, one queue only. Here's what I want:

- Exactly one instance of a queue
- Without transactions
- In a cluster
- With immediate return to the caller after insert

Queued items need to persist beyond a shutdown or crash or something, so the database is used to store them. Since the database is used to store the queue, I need a "process" that wakes up and processes the queue every so often. This has the nice property that the caller just inserts a queue entry, and goes on about its business.

EJB 3.0 has timers. Perfect. Except... they forgot to give you any way to schedule yourself! You gotta be kidding. How about an onStart() callback or something? Yikes. So now I have to cobble something together. All of the (very bad) examples on the Internet rely on a client to start the timer. I don't have a client, thanks. EJB timers are **persisted** to database by default, so once you start them, they don't ever go away, but you can't actually start one??

> *EJB: Look at this great car I built!*
> *Me: Cool, let's drive it!*
> *EJB: Well, it won't actually start... once it's running, it's amazing, but... there's no starter*
> *Me: WTF?*

Anyway. I'm not bitter about having to hack my way around this one. Not a bit.

OK, fine. HUGE hack, I will have the code that inserts into the queue also make sure there is a timer scheduled for the bloody queue processor. Gross.

Oh, and EJB 3.0 doesn't have singletons, but JBoss does (using @Service), so voilá.

But wait, there's more! Two-node cluster... two queue processors! Yay! Now they contend for the queue. So I dig up JBoss's HASingleton stuff, which turns out to be cryptic but effective. I can make my bean depend on a barrier which will make it only deploy on one node at any time. Perfect!

Wait... remember that hack? Now my session bean, which depends on the timer bean, so that it can make sure it's scheduled, which is a gross hack, can't deploy, because the timer bean isn't deployed on all nodes. So the gross hack won't work. I need a grosser hack.

One more bit of background: EJB docs swear up and down not to schedule anything

from the start() event, for some reason I don't recall. Not that start() ever got called for me anyway.

And then it hits me... this new @Depends thing I discovered (for depending on the cluster Barrier)... I can have another bean depend on the timer bean... the timer bean is guaranteed to be started first. It can't schedule itself from its own start() method, but it'll be running when the start() method of the new bean (now called the monitor bean) is called, because that bean @Depends on the timer bean.

I have beaten you, EJB!

Except start() never gets called. Ever. Effing effington. Back to JBoss, which seems to consistently overcome the deficiencies of EJB... my monitor bean can extend org.jboss.system.ServiceMBeanSupport, and it has a startService() method, and it ***actually gets called!*** Amazing.

Deploying this in a cluster, one node runs the timer bean and the monitor bean, in that order. The monitor bean kicks off the timer bean. Perfect. When I shut down JBoss on that node, the same set of events happens on a new node. It works!

So now for some code snippets... FYI I had to make these managed beans to get everything right, it seems. Maybe I could take that out now, I'm not sure. YMMV. You will want to fiddle with which interface you use for JMX vs. the triggering method as well, perhaps.

The timer bean:

```
@Service(objectName="mystuff:service=myTimer")
@Management(MyTimer.class)
@Depends({"jboss.ha:service=HASingletonDeployer,type=Barrier"})
public class MyTimerBean implements MyTimer {
  ...
}
```

The monitor bean:

```
import org.jboss.system.ServiceMBeanSupport;
@Service(objectName="mystuff:service=timerMonitor")
@Management(TimerMonitor.class)
@Depends({"jboss.ha:service=HASingletonDeployer,type=Barrier"})
public class TimerMonitor extends ServiceMBeanSupport {
    @Depends("mystuff:service=myTimer")
    MyTimer timer;
    public void startService() throws Exception {
        timer.trigger();  // method exposed by MyTimer
    }
    ...
}
```

Technorati Tags:
ejb, jboss, cluster, singleton, queue, timer

Posted at 10:16 AM in EJB, JBoss, JEE, Tips and Tricks | Permalink | Comments (0) | TrackBack (0)

0 9 / 0 9 / 2 0 0 9

May 2009
March 2009
February 2009

Subscribe to this blog's feed

MySQL Load Balancing and Read Scalability with EJB and JPA

Well this turned out to be quite an exercise.

The goal: scalable reads with MySQL in master-slave configuration, writing to the master, and reading from N slaves, load balanced in round-robin fashion (or something).

The problem: using JPA (Java Persistence API) instead of direct JDBC calls. Turns out the MySQL ReplicationDriver (used to load balance reads to slaves and send writes to the master) relies on the readOnly state of the Connection in order to decide whether it's a read or a write. With direct JDBC calls, I could get the Connection and toggle the readOnly state as needed.

However, buried under JPA and EntityManager and so on, there's no way to do that. So I looked into other solutions (LBPool, for instance), but nothing out there seems 1) fully baked and 2) intended to do this in a JPA environment.

So I had to do it myself... in digging around in the MySQL Connector/J source, I discovered the loadbalance option, intended for use w/ MySQL Cluster, because in that environment, you don't need to distinguish between master and slave, they're all just nodes.

The loadbalance option wouldn't work directly for me, of course, because I don't want to load balance writes, just reads.

But then it hit me... if I use two data sources, one for reads, and the other for writes, I can stick the read data source behind all my "fetch stuff" APIs, and use the write data source for everything else. Then I can use the loadbalance option for the read data source, because I know for certain I'm never sending it any writes.

Happily, I had modularized and used DAO EJBs, so I just needed a new ReadOnlyDAO, which uses the (injected) EntityManager for the read data source. It also meant I could use the default container transaction behavior, because the loadbalance option relies on transactions to select a host for the connection. Nice.

I hope this helps someone else, because I spent hours and hours digging around the net, and there was very little help to be found.

Technorati Tags:
ejb, java, mysql, jpa, load balancing, scalability

Posted at 04:37 PM in JEE, JPA, Tips and Tricks | Permalink | Comments (0) | TrackBack (0)

0 6 / 1 0 / 2 0 0 9

3J.11: Complexity

So I'm coding along in JEE land and things are going well, EJB 3.0 is relatively sane, JPA makes sense, clustering is easy, I'm making good use of interceptors for useful tasks like authentication, sessions, and caching, and so on.

But then I hit a problem. And of course a 300-line stack trace. But here's the problem with the problem: it's not my problem. It's Hibernate's problem. And that is precisely why I have avoided these behemoth environments in the past... there's a bug, deep in the bowels of everything, and I can no longer implement entities as the fairly well-thought out JPA spec allows. I have to work around the bugs in the system. If the system was small, I could just fix the bug. But it isn't... it would take months or years to understand the implications of fixing this bug a certain way. So all of the benefit of JPA and entity management comes crashing down because it just doesn't work, and thus can't be relied on.

This is the same "way of life" that Microsoft lives and breathes. Lots of complexity, massive amounts of inscrutable code, if it breaks, just reboot and start over. There is some logic to a fine-grained version of that approach, with appropriate supervision (see Erlang for a master course on that). But to approach a whole system that way, and even bake it in, for lack of preventing it, results in the painful systems most people use today.

It yields lazy programmers who do just enough to get by, bad code that doesn't have to be correct, and inexplicable problems no-one can diagnose. The sad thing is that this state of affairs seems to be "OK" with most people creating software. Or they're not even aware it exists.

Complexity is the enemy. Software infrastructure that is inherently complex and removes programmers from the problem is equally the enemy, especially when it is unnecessary. There is no good reason in the world why anyone should feel compelled to develop in JEE. There's nothing inherently wrong with Java (or more precisely, the JVM), but there are real issues with JEE as an approach to software development.

Side note: I would encourage anyone reading this, who is developing for the JVM, to look at Clojure. Clojure is a Lisp variant that natively targets the JVM. It promotes really good software development practices, very much in contrast which much of what the JEE world practices.

Technorati Tags:
java, jee, programming, complexity

Posted at 12:56 PM in Java, JEE | Permalink | Comments (0) | TrackBack (0)

0 5 / 0 2 / 2 0 0 9

3J.10: Closures

When I have this situation:

- Need to call a function, passing the same argument, on all the elements of some collection
- Don't know function or argument in advance

I think lambdas and closures. Unfortunately, Java has neither (one of the language's most significant shortcomings, related to why armies of mediocre programmers use it).

However, I needed closures, so I implemented them, albeit in a pretty restricted way. But it works for what I need because the calling method knows everything, I just don't

want a huge switch statement and I don't want to duplicate the logic. It's just wrong, I shouldn't have to. (mini-rant interlude... languages that claim to promote reuse, refactoring, etc. but don't have first-class functions and closures are just lies)

Here's what I did:

```
import java.lang.reflect.*;

public class Closure {
   private Method method;
   private Object arg;

   Closure(Method meth, Object arg) {
      this.method = meth;
      this.arg = arg;
   }

   public static Closure newClosure(Class object, String methodNa
      try {
         Method meth = object.getDeclaredMethod(methodName, new C
         return new Closure(meth, arg);
      } catch (NoSuchMethodException e) {
         return null;
      }
   }

   public void call(Object obj) throws IllegalAccessException, In
      this.method.invoke(obj, this.arg);
   }
}
```

This could be extended quite a bit of course (maybe w/ subclasses for different closure args, methods for multiple arguments to each call, etc.). Using it is pretty simple:

```
Closure c = Closure.newClosure(Foo.class, "methodName", String.cl
for (Foo o : listOfFoos) {
   c.call(o);
}
```

I've seen other stuff like this around, but I thought it was worth the exercise to do it myself. Obviously there is no actually good solution to this in Java because the language simply lacks the necessary features. But this technique can be pretty handle for saving a lot of code duplication or ugly switching.

Technorati Tags:
java, programming, closures

Posted at 10:51 AM in Java, Patterns | Permalink | Comments (1) | TrackBack (0)

0 3 / 2 4 / 2 0 0 9

3J.9: JAX-RS

We need to implement a REST-ish API and I decided to take a look at JAX-RS (in

RESTEasy form, since we're using JBoss already). In short, I'm really impressed. I usually expect Java frameworks to be big and ugly, but this one is pretty small and simple. It basically does the job of RESTful URL routing via simple annotations (no XML!), URL parameter parsing (again, via annotation), and optionally automatic XML and JSON converting.

Coming from Django and Rails, it's really good to see the Java community embrace some simplicity and elegance in web frameworks. JAX-RS has the benefit of being new and thus not saddled with baggage and legacy support issues.

It isn't perfect, I wish I could annotate the API and *NOT* the data objects for XML conversion. The data objects are used all over the place and it's silly to annotate them for one service where three or four others will totally ignore the annotations.

But I can implement that with my own @Provider... I just wish I didn't have to. :)

In any case, check out RESTEasy, it's good stuff.

Technorati Tags:
java, jee, jaxrs, rest

Posted at 12:18 PM in JAX-RS, JEE, REST | Permalink | Comments (0) | TrackBack (0)

0 3 / 1 1 / 2 0 0 9

3J.8: On Facades

I often see examples of facade patterns in books. Most of the examples I've seen have coarse-grained methods that implement complex logic packaged up into one logical operation. This is great. Most of the examples also pass entities around to and from clients. This is not so great, IMO, at least in some circumstances.

The problem with passing entities around is that it creates a bond between a UI/presentation layer and the actual database architecture, and I'm not sure that's healthy. In fact, I'm pretty sure it isn't, in most cases. This depends somewhat on the nature of the application, of course. If the application consists of several interworking components, operating a process on an entity or set of entities, then passing the entities around makes perfect sense.

On the other hand, if there's a core application implemented behind a facade, with UI/presentation layers, it makes a great deal of sense to shield the presentation layers from the database architecture.

We recently received new requirements (imagine that!) which demanded a different (evolved) database architecture, and we took the opportunity to build a more scalable architecture as well. We did this without changing the existing facade provided to UI and API layers, because we chose initially not to pass entities around, but instead pass model objects which went with the facade.

So while the backend made major relationship changes, the facade presented to the UI never changed. That is, after all, what a facade (in real life) is for, right?

Technorati Tags:
facade, design pattern, ejb, jee

Posted at 11:07 AM in JEE, Ramblings | Permalink | Comments (0) | TrackBack (0)

0 2 / 2 7 / 2 0 0 9

3J.7: Transactions

I knew vaguely that Java (particularly JEE) was famous for transactions, among other things. However, it wasn't until diving in to EJB3 that the power of JEE transactions, especially as handled in EJB3, became clear.

EJB3 supports transactions in multiple ways, but let's take a simple example of default (container-managed) transactions in a remote EJB. If no transaction is specifically created on the client, the container will begin a transaction upon entry to a method in the EJB. That transaction lives for the duration of the call, *and even follows local or remote EJB method invocations*.

Where this gets really powerful is in a session bean presenting a facade. Imagine a web tier and an application tier, each in separate JBoss clusters. The web tier access the application tier via an @Remote EJB facade. The web tier invokes one method on the facade to accomplish a multi-step task, and JBoss begins a new transaction upon entry to that method. Now the method may need to access multiple databases (via @Local or @Remote session beans, implementing DAO patterns, let's say) or call another EJB to perform some operation. *All of those operations* are covered in the transaction. If the last operation breaks, the *entire* multi-step transaction across all of those EJBs will be rolled back.

That is real power, and it makes the facade pattern incredibly obvious and appealing, because the UI/web tier can simply call one method to really do whatever action is needed, at the highest level, and the EJB receiving the call can safely perform operations across databases and other EJBs, without manually building all kinds of complex transaction logic.

That's very compelling to me, and combined with the "annotations plus sensible defaults" approach of EJB3 and JBoss, you don't need to write 1000 lines of finicky XML to make the magic happen. It's actually easy to develop in, which has always been my problem with Java environments ("easy" being relative, there is still a significant learning curve).

Really good stuff!

Technorati Tags:
ejb, jee, transactions, programming

Posted at 08:30 AM in EJB, JEE | Permalink | Comments (0) | TrackBack (0)

0 2 / 2 5 / 2 0 0 9

3J.6: One Good Blog

I have found very, very few really good blogs about Java EE stuff, but there is one standout: Adam Bien. He clearly understands the JEE world, but also applies real developer wisdom to it. Give it a read.

Technorati Tags:
jee, blog

Posted at 08:07 AM in JEE, Weblogs | Permalink | Comments (0) | TrackBack (0)

0 2 / 1 7 / 2 0 0 9

3J.5: Runtime Annotations

One of the coolest things I've discovered while working on this new project is that Java now has annotations, and that the EJB 3.0 framework makes extensive use of them, pretty much making XML glue a thing of the past (finally!). Who knew you could write a complete Java app with no XML?

It took me a bit of digging to really understand the mechanics of annotations, though, so I thought I'd share. First, I won't repeat Sun's annotations overview, so go check that out before continuing here.

What that document didn't convey is that annotations are automatically subclasses of Annotation and also real interfaces. So annotations you retain for runtime use can be queried for the values set in the annotation when a method or object are annotated.

Annotations are an incredibly versatile and powerful way to separate tasks out of APIs, for example. The API methods implement only the logic specific to the API, and an annotation can be used to describe ancillary behavior. Obviously, this requires some framework, but we are almost always either working in a framework or building one (or should be).

There are a couple good examples in my current project: caching and authentication.

I have a stateless facade bean that interacts with some DAO beans internally, and returns results to the UI, or makes changes on its behalf. It would be fantastic if those routines didn't have to manage caching, and could just implement logic. Annotations to the rescue! Each method takes a specific set of arguments, which can be used for form a cache key specific to that method. How cool is that?

So I annotate each method as follows:

```
@CacheResult(name="getSomeStuff", params={1})
public List<String> getSomeStuff(String userName) {
 List<String> list = new ArrayList<String>();
 list.add("an instance of stuff");
 return list;
}
```

And I annotate each method that would invalidate the result, as follows:

```
@InvalidateCache(params={1,2})
public void setSomething(String userName, String ID, String value
 doSetSomething(ID, s);
}
```

And one of the annotations itself (the other is the same without the name method):

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CacheResult {
 int[] params() default {1};
 String name() default "";
}
```

That's it. The rest of the logic is in an Interceptor class (in EJB3). It looks at the method being called, checks for @InvalidateCache, if it finds it, it takes the parameters noted in the annotation and combines them to form a key, which is then used to delete the cached object.

If it finds @CacheResult instead, it forms a key from the name and specified parameters, and looks it up in the cache. If present, it just returns that result without ever invoking the EJB method. If not found, it invokes the method, caches the result (again w/ the key already calculated), and returns it.

Obviously, if the method signatures were all the same, you wouldn't need to be able to control which parameters go into the cache key.

Now, implementing the annotation processing is fairly straightforward with a little reflection (context is an InvocationContext provided by EJB3 to Interceptor classes):

```
import java.lang.reflect.*;
import java.lang.annotation.*;

Object[] params = context.getParameters();
Method m = context.getMethod();
MyBean bean = (MyBean)context.getTarget();

CacheResult c = m.getAnnotation(CacheResult.class);
if (cacheResult != null) {
 return doCacheResult(context, c);
}
```

cacheResult is now an interface to be used like any other, so we can call cacheResult.params() and cacheResult.name() to get the stuff we need for the cache key.

If you're not working in an Interceptor, you might start by finding methods on a class:

```
Method[] methods = p.getClass().getDeclaredMethods();
```

Recently I built a pretty significant project in Python, and introspection ("reflection") played a big part. So one of the first things I was driven to figure out in Java was how to reflect on classes, methods etc. Annotations open a whole new world that was previously non-existent in Java. Good stuff!

Technorati Tags:

java, annotations, ejb

Posted at 07:39 PM in Annotations, Code, Java | Permalink | Comments (0) | TrackBack (0)

0 2 / 1 6 / 2 0 0 9

3J.4: Jars

Jars are just zip files with classes in them. Some of them have one or two XML files in well-defined places. Very simple. Why isn't that emblazoned across java.com somewhere? Hrm?

One of the things that is so off-putting in the Java world is the self-referential dictionary required to understand WTF anyone is talking about. Everything has to have a fancy name, and almost everything has a name that is somehow a play on coffee or beans or kitchen items.

There's a great quote from JBoss in Action: Configuring the JBoss Application Server:

> *Martin Fowler describes the origin of the term POJO in this way: "The term was coined while Rebecca Parsons, Josh MacKenzie and I were preparing for a talk at a conference in September 2000. In the talk we were pointing out the many benefits of encoding business logic into regular Java objects rather than using Entity Beans. We wondered why people were so against using regular objects in their systems and concluded that it was because simple objects lacked a fancy name. So we gave them one, and it's caught on very nicely." - JBoss In Action; Jamae and Johnson; p. 29*

The various flavors of jar files have some similar element of mystery. There are WARs and EARs and EJB-JARs and so on. Since Java seems to be developed almost entirely in fat IDEs like Eclipse and IDEA, the knowledge of what really is going on seems lost, or at least very muted.

So it was merciful relief to discover that jars are 1) simple and 2) sensible. Especially an EJB jar, which requires two files (MANIFEST.MF, which can be basically empty, and ejb-jar.xml, which can be one or two lines long). Here are the MANIFEST.MF file and the ejb-jar.xml file from a current EJB JAR I'm building:

MANIFEST.MF:

```
Manifest-Version: 1.0

Class-Path:


```

ejb-jar.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xm

  <display-name>
```

```
  core-api </display-name>

  </ejb-jar>
```

Pretty simple, no? Both of those files live in the META-INF/ directory parallel to your classes (usually the root dir of your class package hierarchy, actually).

Posted at 11:08 AM in Duh, Java, JEE | Permalink | Comments (0) | TrackBack (0)

Next »